department of computer engineering		**CompOrgALAB** Computer Organization and Architecture	**06**

Group # : ____________________

Date: _11/26/24__________
Rating: __________________

Members			

Experiment #6:
Final Project

I. Learning Outcomes:

At the end of this laboratory experiment, the students should be able to:

a) Develop a system simulating the operations of memory, CPU, and ISA.

II. Materials:

PC/Laptop
Internet
IDE

III. Overview:

Fetch-Decode-Execute cycle

This cycle describes the basic steps that a CPU follows to execute instructions.

Fetch: The CPU retrieves (fetches) the next instruction from the memory. The address of the instruction to be fetched is stored in the program counter (PC).

Decode: The fetched instruction is decoded to determine what operation needs to be performed. This involves interpreting the opcode (operation code) of the instruction.

Execute: The CPU performs the operation specified by the decoded instruction. This could involve arithmetic or logic operations, data transfers, or control operations.

After the execution, the cycle repeats with the program counter being updated to point to the next instruction in memory.

It's important to note that this cycle is the basic building block of the CPU's operation, and it is repeated for each instruction in a program. The specific details of each step can vary depending on the architecture of the CPU.

In this laboratory, students are going to create a software simulator that would show the inner workings of computers with the involvement of ISA, memory, and CPU.

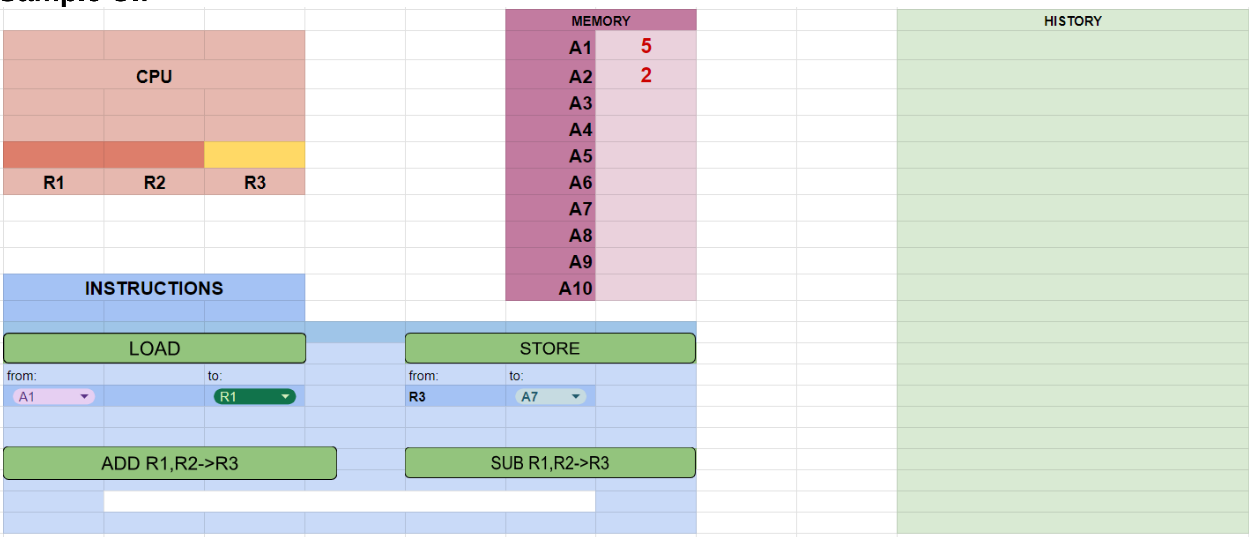
IV. Project Requirements:

- ✓ **Simulator w/ GUI**
- ✓ **System can be built using any programming language, stack, or framework**
- ✓ **Documentation should contain the following:**
  - **Overview: Description/ Features/Purpose**
  - **Discussion of features supported by images**
  - **Discussion of IMPORTANT code supported by code snippets**
  - **Conceptual, Use Case, and Class Diagram**
  - **Answer to the Test Cases**
- ✓ **There will be a live demonstration of the system**
- ✓ **Criteria**
  - **Documentation = 30%**
  - **Code = 20%**
  - **System = 50%**

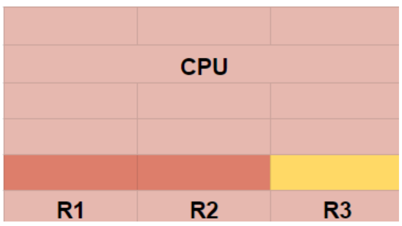
NOTE: You can add features and enhance the GUI by adding animations. Here is the link to the video where this project was inspired [How does a computer run a program? eChalk instructional video - YouTube](#).

Problem Overview
The project will simulate how a computer system executes instructions having three main parts: CPU, Memory, and the Instruction Set.

Sample UI:



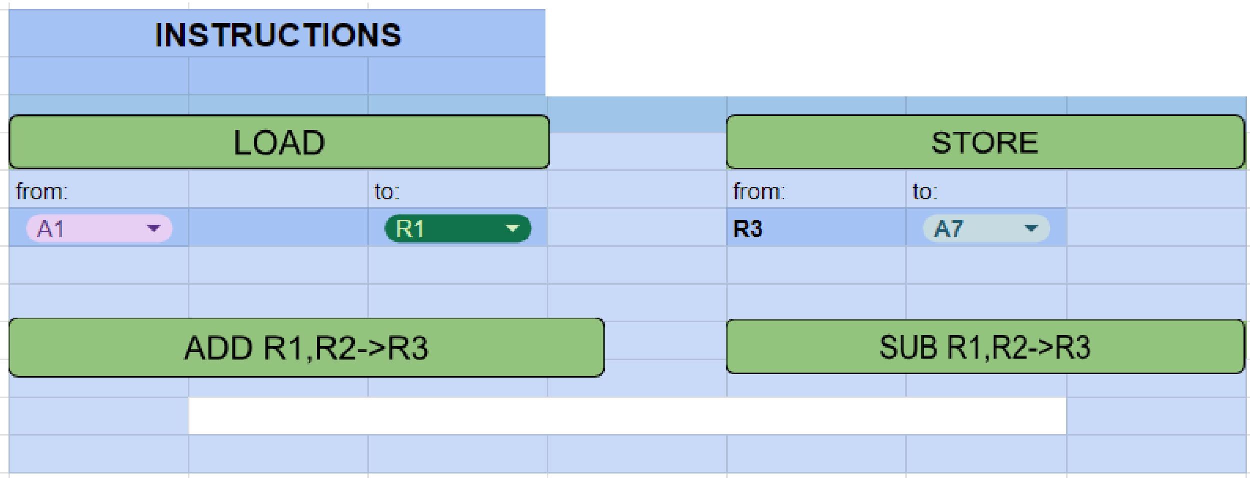
CPU:
The CPU should have 3 registers (R1,R2, and R3). Users should not be able to input anything on the registers. It should only be displaying values as per instruction.



Memory:
The memory should have 10 addresses (A1 to A10). Users should be able to input values.

MEMORY	
A1	5
A2	2
A3	
A4	
A5	
A6	
A7	
A8	
A9	
A10	

Instruction Set:
This is where all the functionalities should be initiated. There should be 4 buttons (the green ones). Each button has its own functionality. For the load() function, it should be able to transfer data stored in the memory (A1-A10) to the registers (R1-R3). Also, after transferring the value, the address where the latter was stored should be emptied. For the store() function, it should transfer value stored in the register to memory. For the add() function, it should add data stored in R1 & R2 and store it in R3. Lastly, for the sub() function, it should subtract data stored in R1 & R2 (R1-R2) and store it in R3. A history should also be added to record the operations.



Computer Organization and Architecture Laboratory
FINAL PROJECT DOCUMENTATION

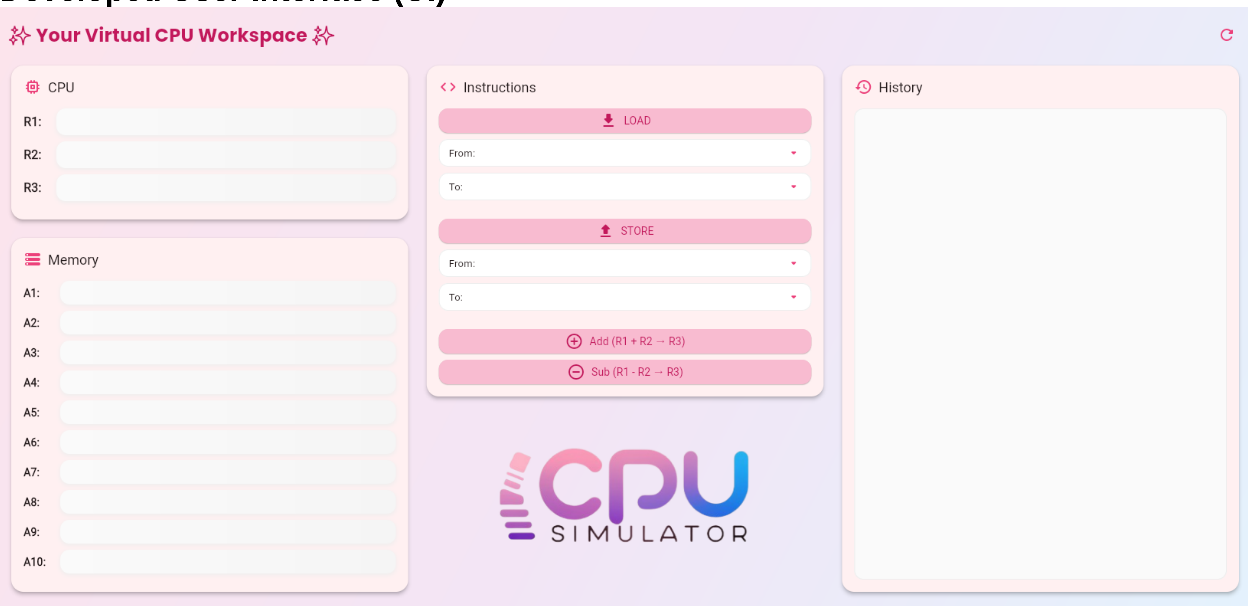
Overview

The CPU Simulator is an innovative application developed using Flutter and Dart, designed to create an engaging and interactive visualization of CPU operations, memory management, and fundamental concepts of instruction set architecture (ISA). This application boasts a modern and animated user interface, enhanced by a visually appealing gradient background that contributes to a responsive design, ensuring a seamless experience across various devices.

Purpose

The primary purpose of the CPU Simulator is to illustrate core concepts of computer architecture through an interactive graphical user interface (GUI). It enables users to visualize the data flow between CPU registers and memory, offering a clearer understanding of how these components interact during processing tasks. Additionally, the simulator provides hands-on experience with basic CPU operations, allowing users to engage directly with the mechanisms that drive computing. A key feature of the application is its ability to log and display operation history in real-time, which not only enriches the learning experience but also allows users to track the performance and behavior of the CPU as they experiment with different operations.

Developed User Interface (UI)



Key Features

- Responsive Flutter UI with adaptive layout for different screen sizes
- Beautiful gradient background and animated components
- Three CPU registers (R1, R2, R3) with real-time value display
- Ten memory addresses (A1-A10) with user input capability
- Basic instruction set (LOAD, STORE, ADD, SUB)
- Comprehensive operation history logging
- Animated transitions and interactions
- Reset functionality

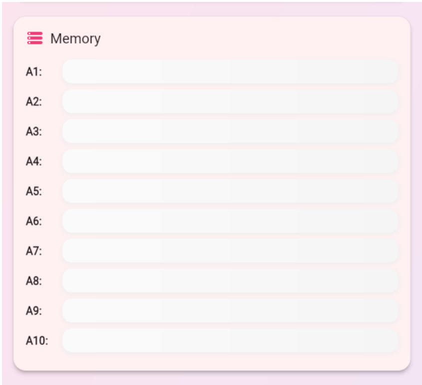
Features Discussion

1. CPU Module

The CPU module features three interactive registers (R1, R2, R3) with:

- Gradient-styled containers for visual appeal
- Click selection functionality
- Real-time value updates
- Visual feedback for selected registers

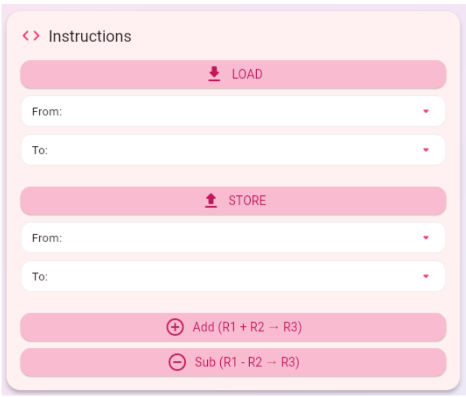
2. Memory Module



The memory module provides:

- Ten addressable locations (A1-A10)
- Interactive text input fields
- Visual selection feedback
- Clear value display
- Input validation for numeric values

3. Instruction Set Module



Features four core operations with animated buttons and dropdown selectors:

LOAD: Transfers data from memory to registers
STORE: Transfers data from registers to memory
ADD: Performs addition (R1 + R2 → R3)
SUB: Performs subtraction (R1 - R2 → R3)

Important Code Snippets

The CPU Simulator was developed using Flutter and Dart, with Visual Studio Code as the primary development environment. Below, you'll find key code snippets that were instrumental in the creation of the system.

- ***Main Widget Structure***

```dart
class CPUSimulator extends StatefulWidget {
  const CPUSimulator({super.key});

  @override
  State<CPUSimulator> createState() => _CPUSimulatorState();
}

class _CPUSimulatorState extends State<CPUSimulator>
    with TickerProviderStateMixin {
  final List<int> registers = List.filled(3, 0);
  final List<TextEditingController> memoryControllers =
      List.generate(10, (index) => TextEditingController());
  final List<String> history = [];
  final ScrollController _scrollController = ScrollController();
```

This code defines the core data structures of the CPU Simulator:
- Creates three CPU registers (R1, R2, R3) to store computational values
- Establishes 10 memory locations (A1-A10) for data storage
- Implements an operation history list to track and display performed actions
- Uses state management to dynamically update and track changes in the simulator

- ***Instruction Operations Implementation***

```dart
void load() {
  if (selectedFromMemory != null && selectedToRegister != null) {
    setState(() {
      int memoryIndex = int.parse(selectedFromMemory!.substring(1)) - 1;
      int registerIndex = int.parse(selectedToRegister!.substring(1)) - 1;
      int value = int.tryParse(memoryControllers[memoryIndex].text) ?? 0;
      registers[registerIndex] = value;
      memoryControllers[memoryIndex].clear();
      _addToHistory(
          "Load: $selectedFromMemory($value) → $selectedToRegister(${registers[registerIndex]})");
    });
  }
}

void store() {
  if (selectedFromRegister != null && selectedToMemory != null) {
    setState(() {
      int registerIndex = int.parse(selectedFromRegister!.substring(1)) - 1;
      int memoryIndex = int.parse(selectedToMemory!.substring(1)) - 1;
      int value = registers[registerIndex];
      memoryControllers[memoryIndex].text = value.toString();
      _addToHistory(
          "Store: $selectedFromRegister($value) → $selectedToMemory($value)");
    });
  }
}

void add() {
  setState(() {
    int r1Value = registers[0];
    int r2Value = registers[1];
    registers[2] = r1Value + r2Value;
    _addToHistory(
        "Add: R1 + R2 → ($r1Value) + ($r2Value) → R3(${registers[2]})");
  });
}

void subtract() {
  setState(() {
    int r1Value = registers[0];
    int r2Value = registers[1];
    registers[2] = r1Value - r2Value;
    _addToHistory(
        "Sub: R1 - R2 → ($r1Value) - ($r2Value) → R3(${registers[2]})");
  });
}
```

LOAD Operation:
- Transfers data from a selected memory address to a specified register
- Checks for valid memory and register selections
- Clears the source memory address after transfer
- Updates the operation history with transfer details

STORE Operation:
- Transfers data from a selected register to a specified memory address
- Validates that the register contains a valid numeric value
- Checks for appropriate memory location selection
- Updates the target memory address with the register's value
- Logs the store operation in the history with relevant details

ADD Operation:
- Performs arithmetic addition of values stored in R1 and R2
- Stores the result in R3 register
- Validates that R1 and R2 contain numeric values
- Adds the operation to the history log with timestamp and calculation details

SUB Operation:
- Performs subtraction by subtracting the value in R2 from R1 (R1 - R2)
- Stores the result in the R3 register
- Validates that R1 and R2 contain numeric values
- Handles potential negative result scenarios
- Adds the subtraction operation to the history log with calculation specifics

- ***Responsive Layout Implementation***

```dart
@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Container(
      decoration: BoxDecoration(
        gradient: LinearGradient(
          begin: Alignment.topLeft,
          end: Alignment.bottomRight,
          colors: [
            Colors.pink.shade50,
            Colors.purple.shade50,
            Colors.blue.shade50,
          ],
        ), // LinearGradient
      ), // BoxDecoration
      child: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: [
            // Header
            Row(
              mainAxisAlignment: MainAxisAlignment.spaceBetween,
              children: [
                Text(
```

```
"✨ Your Virtual CPU Workspace ✨",
style: GoogleFonts.poppins(
  color: Colors.pink.shade700,
  fontWeight: FontWeight.bold,
  fontSize: 24, // Adjust the size as needed
),
).animate().fadeIn(duration: 600.ms).scale(delay: 300.ms), // Text
IconButton(
  onPressed: reset,
  icon: Icon(Icons.refresh_rounded,
    color: Colors.pink.shade400), // Icon
  tooltip: "Reset All",
).animate().fadeIn(duration: 600.ms).scale(delay: 300.ms), // IconButton
],
), // Row
const SizedBox(height: 16),
// Main Content
Expanded(
  child: LayoutBuilder(
    builder: (context, constraints) {
      return constraints.maxWidth > 1200
        ? _buildWideLayout()
        : _buildNarrowLayout();
```

- Implements a responsive user interface that adapts to different screen sizes
- Uses LayoutBuilder to create dynamic layouts
- Arranges CPU registers, memory, and instruction set components flexibly
- Ensures consistent UI experience across various device screen dimensions
- Utilizes column and row widgets for structured, adaptive positioning
- Implements padding and spacing for clean, readable interface design

- ***Animation Implementation***

```
late AnimationController _animationController;
late Animation<double> _scaleAnimation;

@override
void initState() {
  super.initState();
  _animationController = AnimationController(
    duration: const Duration(milliseconds: 300),
    vsync: this,
  ); // AnimationController
  _scaleAnimation = Tween<double>(begin: 1.0, end: 1.1).animate(
    CurvedAnimation(parent: _animationController, curve: Curves.easeInOut),
  );
}
```

- Integrates smooth animations using flutter_animate package
- Adds visual feedback and interactivity to UI components
- Implements fade, scale, and slide animations for buttons and history list
- Creates engaging transitions between different states of the simulator
- Enhances user experience through subtle, meaningful animated interactions

Implementation Details

Flutter packages used in the application:

- flutter_animate for smooth animations
- google_fonts for typography
- Built-in Flutter animation controllers for custom animations

Key UI Components:

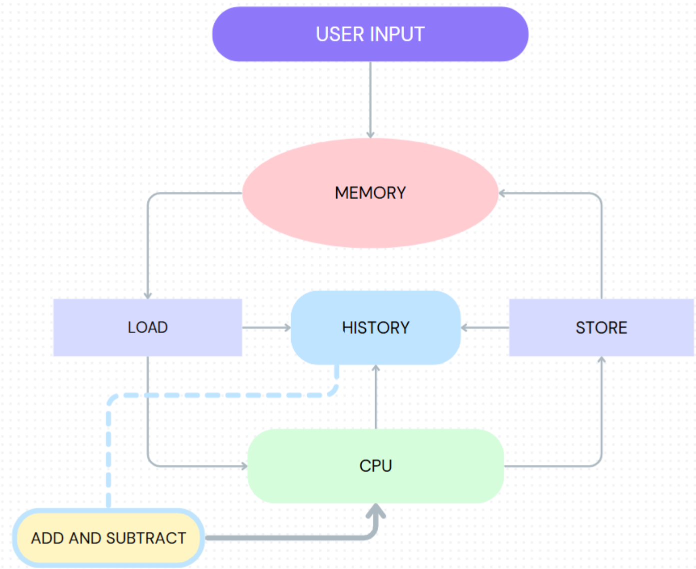
- Gradient background using LinearGradient
- Custom animated buttons using ElevatedButton.icon
- Responsive layout using LayoutBuilder
- Custom styled text fields for memory input

- Animated history list with ListView.builder
- Custom dropdowns for instruction selection

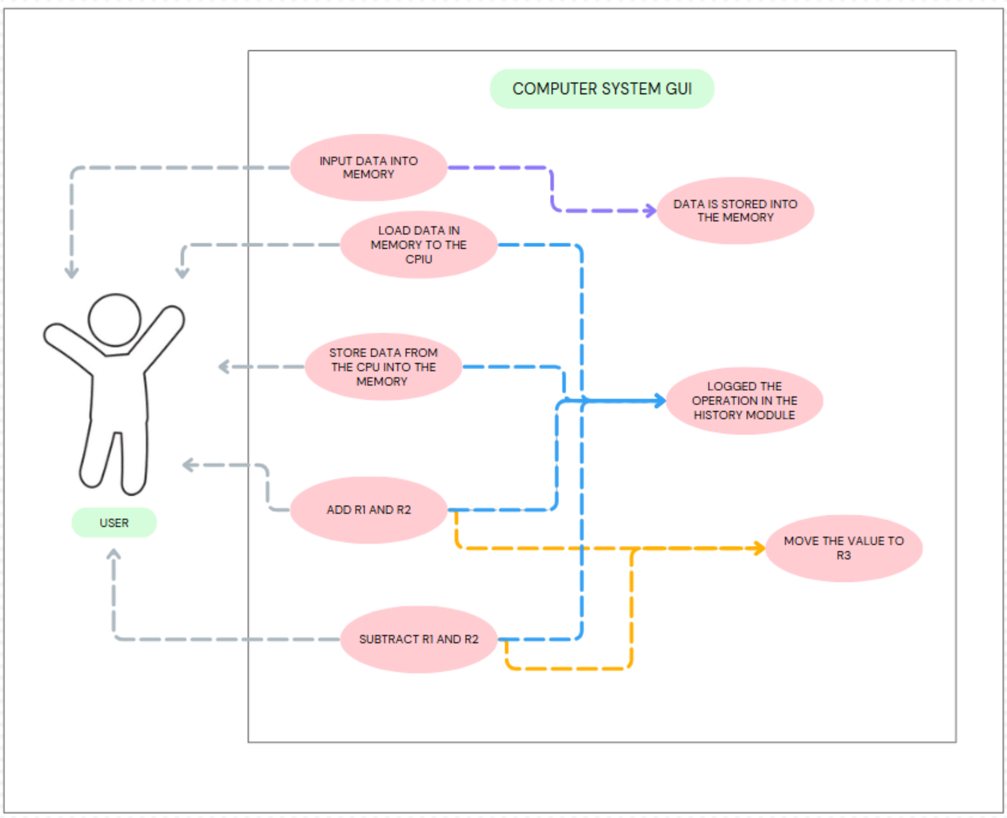
Performance Considerations:

- Efficient state management using setState
- Proper disposal of controllers in dispose method
- Optimized animations using TickerProviderStateMixin
- Memory-efficient list implementations

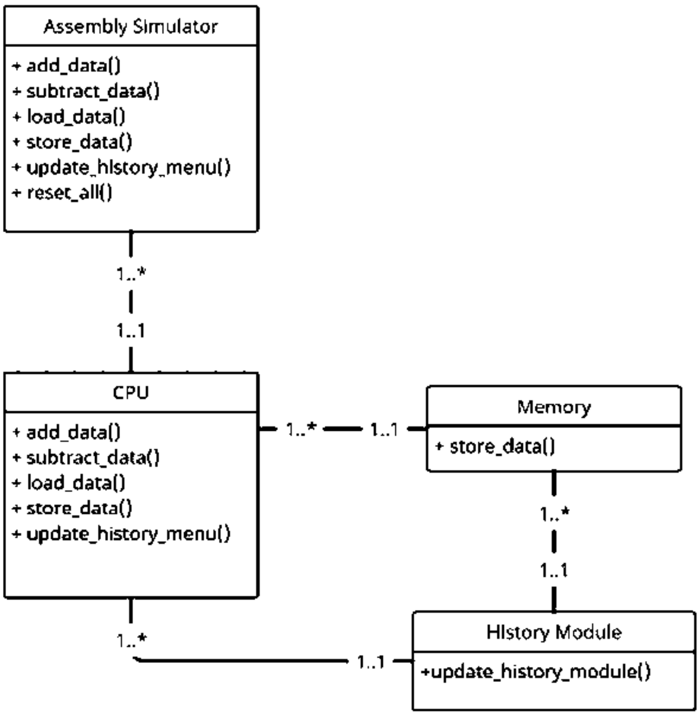
Conceptual Diagram



Use-Case Diagram



Class Diagram



Answers to Test Cases

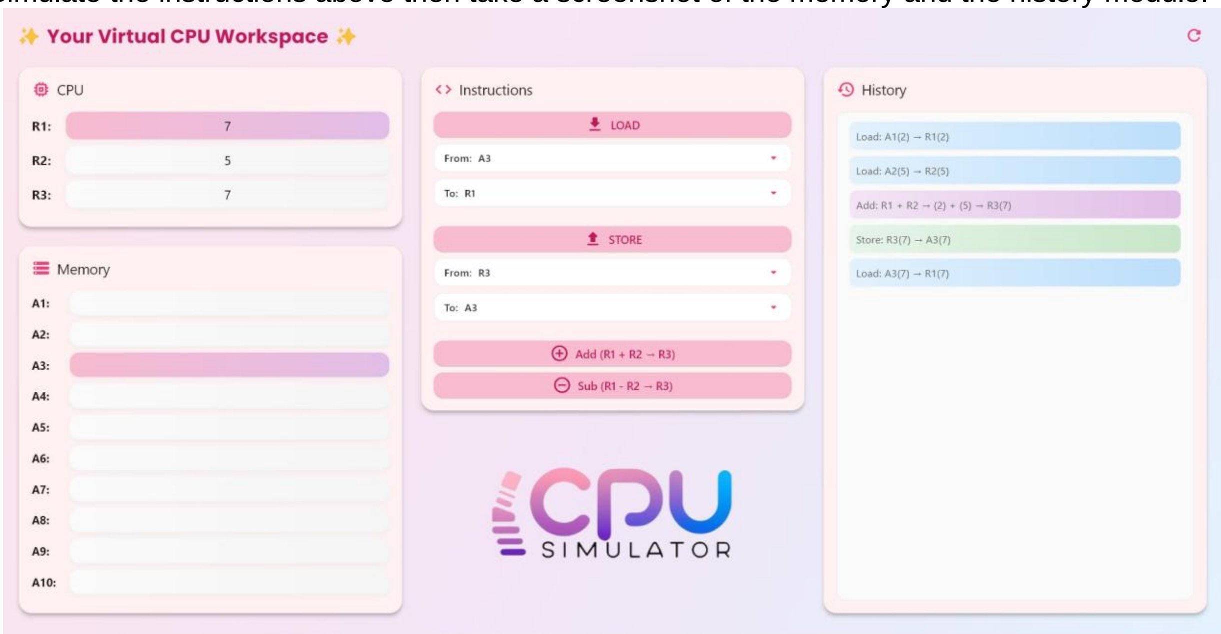
After creating the simulator. Answer the following questions:

1. Suppose the memory has the following contents:

A1	A2
2	5

MOV R1, A1
MOV R2, A2
ADD R1, R2
MOV R3, R1
MOV A4, R3

Simulate the instructions above then take a screenshot of the memory and the history module.



2. Suppose the memory has the following contents:

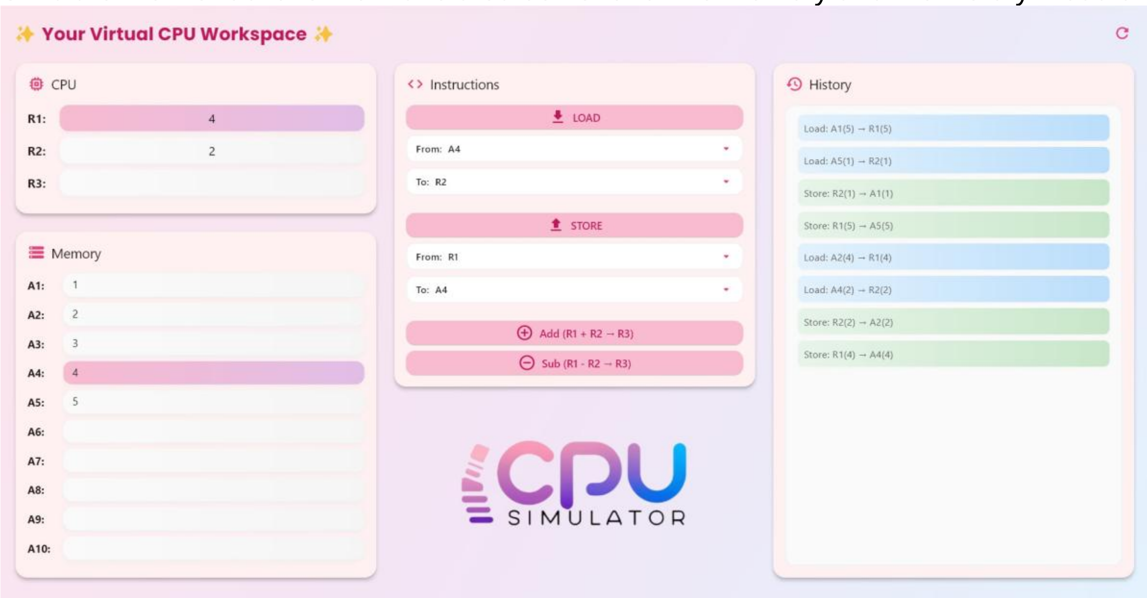
A1	A2	A3	A4	A5
5	4	3	2	1

Make the instructions that would reverse the order of values as follows:

A1	A2	A3	A4	A5
1	2	3	4	5

```
MOV R1, A1
MOV R2, A5
MOV A1, R2
MOV A5, R1
MOV R1, A2
MOV R2, A4
MOV A2, R2
MOV A4, R1
```

Simulate the instructions then take a screen shot of the memory and the history module.



3. Suppose the memory has the following contents:

A1	A3	A6	A7	A10
25	5	30	36	20

Instructions:
MOV R1, A1
MOV R2, A3
SUB R2, R1
MOV R3, R2
MOV A3, R3
MOV R1, A6
MOV R2, A10
ADD R1, R2
MOV R3, R1
MOV A10, R3

Simulate the instructions above then take a screen shot of the memory and the history module.


Your Virtual CPU Workspace
CPU
R1: 30
R2: 20
R3: 50
Memory
A1:
A2:
A3: -20
A4:
A5:
A6:
A7: 36
A8:
A9:
A10: 50
Instructions
LOAD
From: A10
To: R2
STORE
From: R3
To: A10
Add (R1 + R2 → R3)
Sub (R1 - R2 → R3)
CPU
SIMULATOR
History
Load: A1(25) → R1(25)
Load: A3(5) → R2(5)
Store: R1(25) → A1(25)
Store: R2(5) → A2(5)
Load: A1(25) → R2(25)
Load: A2(5) → R1(5)
Sub: R1 - R2 → (5) - (25) → R3(-20)
Store: R3(-20) → A3(-20)
Load: A6(30) → R1(30)
Load: A10(20) → R2(20)
Add: R1 + R2 → (30) + (20) → R3(50)
Store: R3(50) → A10(50)